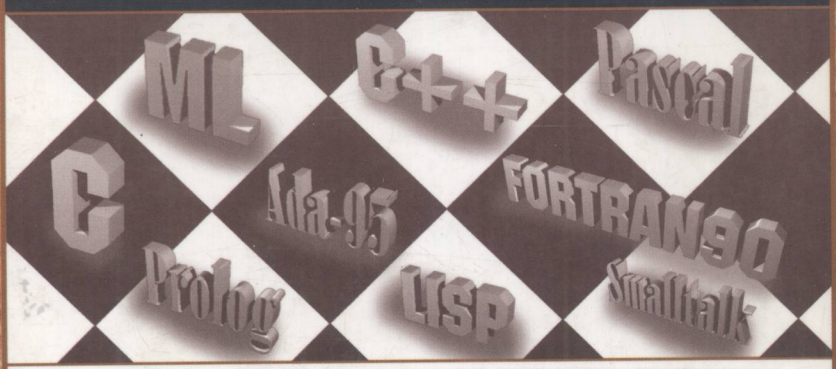
大学计算机教育丛书（影印版）

THIRD EDITION

PROGRAMMING LANGUAGES

DESIGN AND IMPLEMENTATION



程序设计语言设计与实现

第3版

TERRENCE W. PRATT
MARVIN V. ZELKOWITZ

清华大学出版社 · PRENTICE HALL
http://www.tup.tsinghua.edu.cn

PROGRAMMING LANGUAGES

Design and Implementation

THIRD EDITION

程序设计语言

设计与实现

第3版

Terrence W. Pratt

Center of Excellence in Space Data and Information Sciences
NASA Goddard Space Flight Center, Greenbelt, MD

Marvin V. Zelkowitz

Department of Computer Science and
Institute for Advanced Computer Studies
University of Maryland, College Park, MD

清华大学出版社

Prentice-Hall International, Inc.

(京) 新登字 158 号

Programming languages: design and implementation 3rd ed./Terrence W. Pratt and Marvin V. Zelkowitz.

北京市版权局著作权合同登记号: 01-98-0263

图书在版编目(CIP)数据

程序设计语言: 设计与实现: 第 3 版: 英文/普拉特(Pratt, T.W.), 泽尔柯维支(Zelkowitz, M.V.)著. - 影印版. - 北京: 清华大学出版社, 1998.5
(大学计算机教育丛书)
ISBN 7-302-02833-8

Ⅰ. 编… Ⅱ. ①普… ②泽… Ⅲ. 程序语言-英文 Ⅳ. TP312

中国版本图书馆 CIP 数据核字(98)第 01690 号

出版者: 清华大学出版社(北京清华大学校内, 邮编 100084)
因特网地址: www.tup.tsinghua.edu.cn
印刷者: 清华大学印刷厂
发行者: 新华书店总店北京科技发行所
开　本: 850×1168 1/32 **印张:** 21.125
版　次: 1998 年 5 月第 1 版 1998 年 5 月第 1 次印刷
书　号: ISBN 7-302-02833-8/TP·1490
印　数: 0001～5000
定　价: 32.00 元

Preface

This third edition of *Programming Languages: Design and Implementation* continues the tradition developed in the first two editions to describe programming language design by means of the underlying software and hardware architecture that is required for execution of programs written in those languages. This provides the programmer with the ability to develop software that is both correct and efficient in execution. In this new edition, we continue this approach, as well as improve upon the presentation of the underlying theory and formal models that form the basis for the decisions made in creating those languages.

Programming language design is still a very active pursuit in the computer science community as languages are "born," "age," and eventually "die." This third edition represents the vital languages of the mid-1990s. Chapters on COBOL, PL/I, SNOBOL4, and APL have been dropped. Discussions on C, C++ , ML, Prolog, and Smalltalk have been added to reflect the evolution of programming language design and the emergence of new paradigms within the community. Pascal is starting to age, and Ada and FORTRAN have been renewed with new standards, Ada 95 and FORTRAN 90, respectively. It is interesting to speculate as to whether any of these languages will be in future editions of books such as this one. For both of us, the deletion of SNOBOL4 was a considerable loss. It is one of the most interesting and powerful languages ever developed, although it still lives on as "shareware" in PCs.

At the University of Maryland, a course has been taught for the past 20 years that conforms to the structure of this book. For our junior-level course, we assume the student already knows Pascal and C from earlier courses. We then emphasize ML, Prolog, C++ , and LISP, as well as include further discussions of the implementation aspects of C and Pascal. The study of C++ furthers the students' knowledge of procedural languages with the addition of object oriented classes, and the inclusion of LISP, Prolog, and ML provide for discussions of different programming paradigms. Replacement of one or two of these by FORTRAN, Ada, or Smalltalk would also be appropriate.

It is assumed that the reader is familiar with at least one procedural language, generally C, FORTRAN, or Pascal. For those institutions using this book at a lower level, or for others wishing to review prerequisite material to provide a framework for

discussing programming language design issues, Chapters 1 and 2 provide a review of material needed to understand later chapters. Chapter 1 is a general introduction to programming languages, while Chapter 2 is a brief overview of requirements for programming languages.

The theme of this book is language design and implementation issues. Part I forms the core of an undergraduate course in programming languages. Chapters 3 through 8 are the basis for this course by describing the underlying grammatical model for programming languages and their compilers (Chapter 3), elementary data types (Chapter 4), encapsulation (Chapter 5), statements (Chapter 6), procedure invocation (Chapter 7), and inheritance (Chapter 8), which are the central concerns in language design. Examples of these features are described in a variety of languages and typical implementation strategies are discussed.

The topics in this book cover the 12 knowledge units recommended by the 1991 ACM/IEEE Computer Society Joint Curriculum Task Force for the programming languages subject area [TUCKER et al. 1991]. For institutions using this book at a higher level or those wishing to address more advanced topics, Chapter 9 continues the discussion of parsing that is first introduced in Chapter 3 and brings in the concept of programming language semantics with discussions of program verification, denotational semantics, and the lambda calculus with an introduction to undecidability and NP completeness. This provides the reader with an overview of more advanced courses in the programming language, software engineering, and computational theory areas of computer science. For this material, prior experience with the predicate calculus and mathematical logic would help. In addition, Chapter 9 addresses current issues in parallel programming, provides an introduction to current research in hardware and software, and suggests what are likely to be the programming language design issues in the future.

While compiler writing was at one time a central course in the computer science curriculum, there is increasing belief that not every computer science student needs to be able to develop a compiler; such technology should be left to the compiler specialist, and the "hole" in the schedule produced by deleting such a course might be better utilized with courses such as software engineering, database engineering, or other practical use of computer science technology. However, we believe that aspects of compiler design should be part of the background for all good programmers. Therefore, a focus of this book is how various language structures are compiled, and Chapter 3 provides a fairly complete summary of parsing issues.

The nine chapters of Part I emphasize programming language examples in FORTRAN, Ada, C, Pascal, ML, LISP, Prolog, C++, and Smalltalk. Additional examples are given in PL/I, SNOBOL4, APL, BASIC, and COBOL, as the need arises. The sections of Part II, however, are organized around individual languages. Each section describes a different language and shows how that language provides the features described in the first nine chapters of Part I. The goal is to present each language as a consistent implementation of the software architecture given in the first half of the book. While certainly not a reference manual for each language, each section should provide enough information for the student to solve interesting

class problems in each of those languages without the need to purchase separate language reference manuals. (However, having a few of those around for your local implementation is certainly a big help.)

While discussing all of the languages briefly during the semester is appropriate, we do not suggest that the programming parts of this course consist of problems in each of these languages. We think that would be too superficial in one course. Nine programs in nine different languages would be quite a chore and provide the student with little in-depth knowledge of any of these languages. We assume that each instructor will choose three or four of the Part II languages and emphasize those.

All examples in this book, except for the most trivial, were tested on an appropriate translator; however, as we clearly point out in Section 1.3.3, correct execution on our local system is no guarantee that the translator is processing programs according to the language standard. We are sure that Mr. Murphy is at work here, and some of the "trivial" examples may have errors. If so, we apologize for any problems that may cause.

To summarize, our goal in producing this third edition was to:

- Provide an overview of the key paradigms used in developing modern programming languages;
- Highlight several languages, which provide those features, in sufficient detail to permit programs to be written in each language demonstrating those features;
- Explore the implementation of each language in sufficient detail to provide the programmer an understanding of the relationship between a source program and its execution behavior;
- Provide sufficient formal theory to show where programming language design fits within the general computer science research agenda; and
- Provide a sufficient set of problems and alternative references to allow students the opportunity to extend their knowledge of this important topic.

We gratefully acknowledge the valuable comments received from Henry Bauer, Hikyoo Koh, John Mauney, and Andrew Oldroyd on earlier drafts of this manuscript and from the 118 students of CMSC 330 at the University of Maryland during the Spring, 1995 semester who provided valuable feedback on improving the presentation contained in this book.

Perhaps 70% of the text has been rewritten between edition 2 and edition 3. We believe the new edition is a considerable improvement over the previous version of this book. We hope that you agree.

Terry Pratt
Greenbelt, Maryland

Marv Zelkowitz
College Park, Maryland

Contents

Part I

Concepts

In Part I we study the features that make up programming languages. We address the following issues: What are the features that form a programming language? How do they interact? How are they implemented? What are the various paradigms that describe program execution? We give examples in several languages that demonstrate answers to each of these questions.

Later in Part II, we look at each language individually and describe how that particular language addresses the above questions.

Chapter 1

The Study of Programming Languages

Any notation for the description of algorithms and data structures may be termed a programming language; however, in this book we are mostly interested in those that are implemented on a computer. The sense in which a language may be "implemented" is considered in the next two chapters. In the remainder of Part I the design and implementation of the various components of a language are considered in detail. The goal is to look at language features, independent of any particular language, and give examples from a wide class of commonly used languages.

In Part II of this book, we illustrate the application of these concepts in the design of nine major programming languages and their dialects: Ada, C, C++, FORTRAN, LISP, ML, Pascal, Prolog, and Smalltalk. In addition, we also give brief summaries about other languages that have made an impact on the field. This list includes APL, BASIC, COBOL, Forth, PL/I, and SNOBOL4. Before approaching the general study of programming languages, however, it is worth understanding why there is value in such a study to a computer programmer.

1.1 WHY STUDY PROGRAMMING LANGUAGES?

Hundreds of different programming languages have been designed and implemented. Even in 1969, Sammet [SAMMET 1969] listed 120 that were fairly widely used, and many others have been developed since then. Most programmers, however, never venture to use more than a few languages, and many confine their programming entirely to one or two. In fact, practicing programmers often work at computer installations where use of a particular language such as C, Ada, or FORTRAN is required. What is to be gained, then, by study of a variety of different languages that one is unlikely ever to use?

There are excellent reasons for such a study, provided that you go beneath the superficial consideration of the "features" of languages and delve into the underlying design concepts and their effect on language implementation. Six primary reasons

come immediately to mind:

1. *To improve your ability to develop effective algorithms.* Many languages provide features that when used properly are of benefit to the programmer but when used improperly may waste large amounts of computer time or lead the programmer into time-consuming logical errors. Even a programmer who has used a language for years may not understand all of its features. A typical example is *recursion,* a handy programming feature that when properly used allows the direct implementation of elegant and efficient algorithms. But used improperly, it may cause an astronomical increase in execution time. The programmer who knows nothing of the design questions and implementation difficulties that recursion implies is likely to shy away from this somewhat mysterious construct. However, a basic knowledge of its principles and implementation techniques allows the programmer to understand the relative cost of recursion in a particular language and from this understanding to determine whether its use is warranted in a particular programming situation.

   New programming methods are constantly being introduced in the literature. The best use of concepts like object-oriented programming, logic programming, or concurrent programming, for example, requires an understanding of languages that implement these concepts.

2. *To improve your use of your existing programming language.* By understanding how features in your language are implemented, you greatly increase your ability to write efficient programs. For example, understanding how data such as arrays, strings, lists, or records are created and manipulated by your language, knowing the implementation details of recursion, or understanding how object classes are built allows you to build more efficient programs consisting of such components.

3. *To increase your vocabulary of useful programming constructs.* Language serves both as an aid and a constraint to thinking. People use language to express thoughts, but language serves also to structure how one thinks, to the extent that it is difficult to think in ways that allow no direct expression in words. Familiarity with a single programming language tends to have a similar constraining effect. In searching for data and program structures suitable to the solution of a problem, one tends to think only of structures that are immediately expressible in the languages with which one is familiar. By studying the constructs provided by a wide range of languages, and the manner in which these constructs are implemented, a programmer increases his programming "vocabulary." The understanding of implementation techniques is particularly important, because in order to use a construct while programming in a language that does not provide it directly, the programmer must provide his own implementation of the construct in terms of the primitive elements actually provided by the language. For example, the subprogram control structure known as *coroutines* is useful in many programs,